On the Interoperability of DEVS components: On-Line vs. Off-Line Strategies

Luc Touraille, Mamadou Kaba Traoré, David R.C. Hill

On the Interoperability of DEVS Components

On-Line vs. Off-Line strategies

Luc Touraille[1], Mamadou K. Traoré[2], David R.C. Hill[3]

Research Report LIMOS/RR-09-04

12 juin 2009

[1] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, touraille@isima.fr
[2] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, traore@isima.fr
[3] LIMOS-ISIMA, Campus des Cézeaux, 63177 Aubière, drch@isima.fr

Abstract

During the last years, the DEVS community provides many contributions towards the realization of a world-wide platform for collaborative Modeling & Simulation. The goal of such a platform would be to enable the sharing and reuse of models between scientists, as well as the seamless simulation of distributed and heterogeneous models. Therefore, one of the major research fields is the definition of architectures for integrating heterogeneous DEVS components, meaning simulators and/or models written in different frameworks and programming languages. In this work, we present three different strategies for providing such interoperability between DEVS components. The first focuses on standardizing exchanges between simulators, and has been explored in previous works. The two others strategies are more prospective; in keeping with Model-Driven Engineering, they place the model at the center of their architecture and make extensive use of model transformations. To make this possible, we defined a platform and language-independent format for describing and sharing DEVS models, called DEVS Markup Language.

Keywords: DEVS, interoperability, web service, XML

Résumé

Depuis plusieurs années, la communauté DEVS travaille à la réalisation d'une plate-forme collaborative de Modélisation et de Simulation. Celle-ci devrait permettre aux scientifiques de par le monde de partager leurs modèles et de réutiliser ceux déjà existants, mais également d'exécuter facilement des simulations faisant intervenir des modèles distribués et hétérogènes. Par conséquent, beaucoup d'efforts sont menés pour définir des architectures permettant d'intégrer des composants DEVS hétérogènes, c'est-à-dire des simulateurs et/ou des modèles développés dans des cadriciels et des langages de programmation différents. Dans cet article, nous présentons trois stratégies possibles pour rendre possible l'interopérabilité entre composants DEVS. La première se base sur la standardisation des échanges entre simulateurs, et a été explorée dans des travaux précédents. Les deux autres sont plus prospectives, et s'inscrivent dans l'Ingénierie Dirigée par les Modèles. En effet, les modèles sont au centre de leurs architectures et l'interopérabilité se fait par le biais de nombreuses transformations de modèle. Pour rendre ces deux approches possibles, nous avons défini un format de représentation des modèles DEVS, appelé DEVS Markup Language, qui se veut totalement indépendant de toute plate-forme et langage de programmation.

Mots-clés : DEVS, interopérabilité, service web, XML

1. Introduction

One of the current trends in software engineering is to develop applications not as monolithic entities like they were a few years back, but as a collection of loosely-coupled/highly-cohesive components that can be easily combined, substituted and reused, named services. In addition to the notion of "Software As A Service" introduced a decade ago [Keith at al. 2000] and now knowing business success, the Service Oriented Architecture (SOA) is now commonly implemented through web services, a widely used technology based on several accepted standards [Newcomer and Lomow 2005]. Among the advantages of web services, we retain the fact that they facilitate the integration of heterogeneous components, notably through the standardization of interfaces description (Web Service Description Language), messages definition (XML Schema) and communication protocol (Service Object Access Protocol). Therefore, we can leverage this technology to provide straight-forward interoperability of DEVS components [Zeigler et al. 2000].

Our goal is to provide a simple and mostly automated way of executing simulations that involve distant and/or heterogeneous[4] DEVS models. This can be achieved by taking two different approaches: simulator-based interoperability and model-based interoperability. Both these approaches need a standardized, platform-independent representation of DEVS models. We defined the DEVS Markup Language, an XML-based language for describing and sharing DEVS models.

2. Simulator-based interoperability

The main idea of this approach, as used in DEVS/SOA [Seo 2009], is to have a collection of simulation services distributed over the internet. These services provides several operations for simulating atomic or coupled DEVS models in a unified manner, by using the DEVS simulation protocol and the closure under coupling property of coupled models to make them appear as atomic model. The overall simulation is coordinated by a main service, which acts like an entry point for the user. This architecture is summarized in the Figure 1.

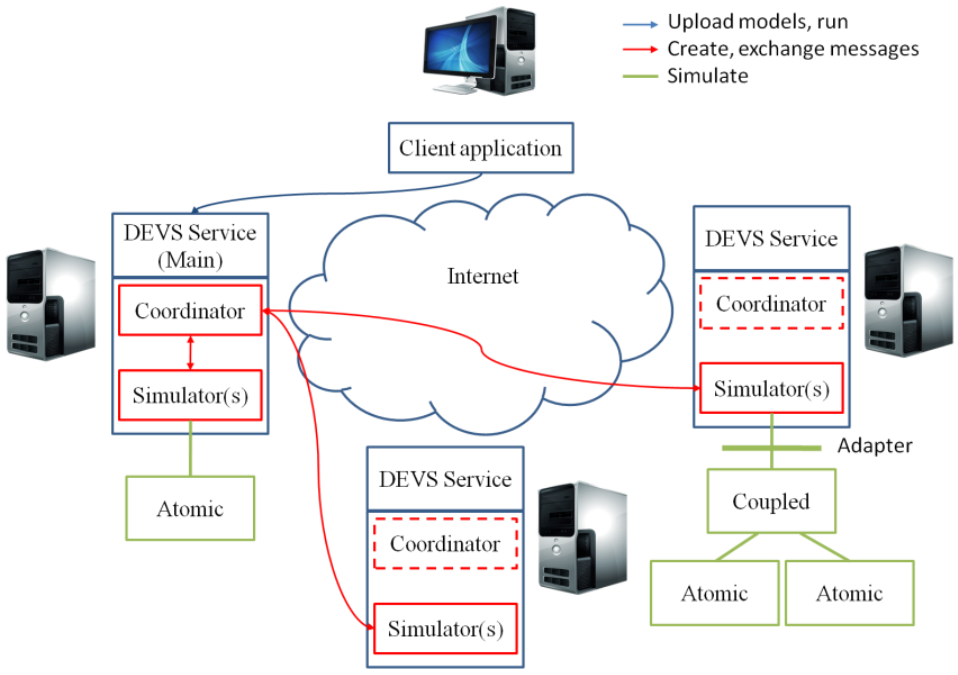

Figure 1: DEVS components interoperability through simulators communication

In order to communicate, the DEVS services expose a standard interface that accepts the usual simulation protocol messages: initialize, get time of next event, run transition and so on.

[4]By heterogeneous we mean written in different programming languages and/or for different DEVS frameworks.

From a user perspective, the simulation process consists in:

1. writing a coupled DEVS model, using its favorite framework.
2. providing a list of DEVS servers on which he wishes to distribute the simulation.
3. deploying DEVS models to the servers (either by downloading them from his location or activating existing remote models)
4. running the simulation

3. Model-based interoperability

This second type of interoperability is based on the fact that most of the time modelers already have or can easily install a DEVS simulator on their machine. However, it is much more difficult for them to retrieve and reuse models already existing on other scientists' computers. Therefore, we need some way to provide access to and share these numerous models.

In this perspective, we distinguish between *on-line* (or dynamic) and *off-line* (or static) interoperability, which are both centered on the models, and bring into play several methods related to Model Driven Engineering (MDE) [Schmidt 2006].

3.1. On-line (dynamic) model-based interoperability

In this solution, models themselves are deployed as web services instead of simulators (see **Erreur ! Source du renvoi introuvable.**). Using this model driven approach, the operations invoked through the network are no longer simulation mechanisms, but model functions, such as δ_{int}, δ_{ext}, the time advance function, etc.

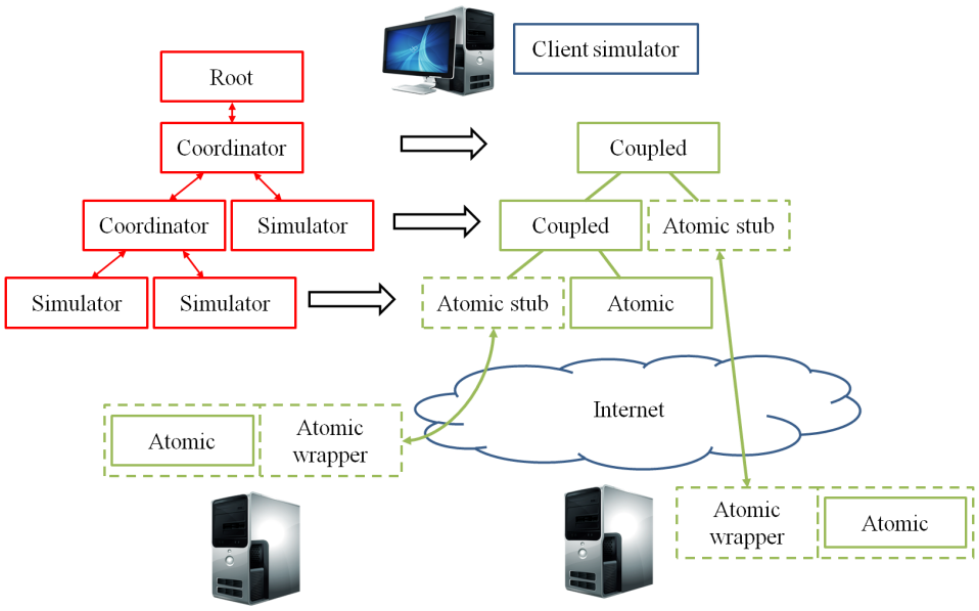

Figure 2: Local simulation of distant and local models

Given a model written for a specific framework, we need to generate its representation in a platform-independent language, such as the DEVS Markup Language (DML) [Touraille et al. 2009], described below, or the DEVS Modeling Language (DEVSML) [Mittal et al. 2007]. This description will then be used to generate:

- the web service description, as a WSDL file
- the wrapper that will adapt the model interface to the service interface, by forwarding the operation calls.

Once this is done, the client can retrieve the description of the model, and generate a stub that will conforms to its framework and behave exactly as the original model, by invoking the wrapper service. The automatic generations cited above can be seen as automatic transformations in the model driven engineering terminology [Schmidt 2006]. Regarding distant coupled models, two solutions are possible. The first is to use some kind of adapter, as in the previous section, which will make the coupled model appear as an atomic one. The second possibility is to take into account that coupled model don't have any operational semantics. All they do is describing their ports and their couplings, in a static way. As a consequence, we can use their DML representation to generate a copy of the model in the local framework, without losing any useful information.

In a nutshell, the model provider:

1. writes a model in his favorite DEVS framework.
2. automatically generates the model's DML description, from which he generates a WSDL file and a web service encapsulating the model.
3. deploys the web service, providing access to the underlying model.

On his side, all the model consumer has to do when writing a coupled model in a given framework is:

1. find the location of the models he needs, using some kind of model directory or more classical discovery procedures such as web searches or acquaintances.
2. give the addresses of the models' web services to a generation tool that will download the WSDL files and generate client stubs adhering to the modeler framework.
3. run the simulation. The framework's simulator deals with local and distant models without even knowing it.

3.2. Off-line (static) model-based interoperability

On the previous approach, we used DML description of models to generate client and server stubs; however, DML includes not only the interface of models, but also their dynamics, described in a platform and language independent format (see Section 3). Consequently, we can go one step further in automatic model transformations and generate entire local copies of distant models.

Using transformations based on code parsing, we can generate the DML description of existing models written for a specific framework. These DML files can be stored in *model repositories*, accessible through Internet, so that anyone (with the proper authorizations) can access them. When facing a problem, the modeler starts by searching an existing model that could fulfill his requirements. If there is none, he can write his own but still take advantage of existing works by reusing them in his coupled models. To do so, the user download the model description from its repository then use generic transformations to generate the code corresponding to the model in his framework. He ends up simulating his models and those of other scientists on his own simulating platform.

Compared to on-line interoperability, this approach has the benefit that it drastically reduces the communications with remote servers. In fact, after the model description has been downloaded, there is no more need for accessing the network (hence the "off-line" appellation). However, the drawback of this is that a given model can have several unsynchronized versions all over the world. A solution to this issue could be to use some sort of update feed, but this is still more complicated than simply invoking a service without caring about the underlying implementation, which can continuously improve and evolve.

To sum up both interoperability solutions we exposed, we can say that the simulator-based approach is more aimed at coupling distributed or local models over distant servers, whereas the model-based approach aimed at integrating distributed models in a local simulation.

4. DML, a markup language for the sharing and interoperability of DEVS models

As stated in the previous sections, many solutions for providing seamless interoperability of DEVS models involve a standardized language for describing these models. The DEVS Markup Language (DML) [Touraille et al. 2009] is an attempt to unify previous propositions for such a language [Janoušek et al. 2006] [Mittal et al. 2007] [Risco-Martín et al. 2007], while at the same time making it as generic and flexible as possible.

4.1. Introduction

An urgent necessity in the Modeling and Simulation domain is to define standards to make for interoperability and reusability of models and simulators. Thanks to these standards, a modeler should be able to seamlessly execute the following operations:

- store model specifications and data trajectories for further use.
- perform reasoning on these data, e.g. select a model in a library based on precise criteria, or execute analysis on a model structure or a generated output trajectory.
- share models among the community, to enhance productivity and quality by reusing well-tested and validated models.
- visualize simulation results through different interfaces (charts, animation), and graphically create/edit models.
- transform model specifications into executable source code for several target simulators, and perform simulation.

Due to the lack of standard, these use cases are currently rather difficult to fulfill. Most actual models are described either in natural language, problematic – if not impossible – to process with a computer, or in a specific programming language, using types and functions peculiar to one simulator. This heterogeneity obstructs the collaboration between modelers, and makes impossible the development of generic tools that could be used by the entire community.

To solve this problem, we propose a simulator-independent language to describe models formalized in the Discrete Event System Specification (DEVS) [Zeigler et al. 2000]. This language, built on eXtensible Markup Language (XML) [Abiteboul et al. 1999], must allow the specification of all the aspects needed to realize the previously exposed use cases.

The next section presents the works related to the definition of an XML representation of DEVS models. Section 3 exposes several limits of these previous propositions, and propounds some ideas that are further developed in section 4. Finally, we synthesize our results and present future work in section 5.

4.2. Related works

The DEVS Standardization group pinned down four areas of the DEVS framework needing standardization [Vangheluwe et al. 2001]:

- the DEVS formalism in itself, and its variants
- model representation, which should describe the model structure and dynamics in a platform-independent manner
- minimum requirements for a simulator to be labeled "DEVS-compliant"
- model libraries, aimed at providing a collection of models usable out of the box

This paper focuses on the second point, namely the standardization of model representation. To achieve this goal, the choice of XML seems obvious; indeed, a consensus emerged among the M&S community about XML being the best tool to describe models in a standardized way [Brutzman et al. 2002] [Fishwick 2002]. This general agreement is due to numerous reasons, such as the wide use of XML in Service Oriented Architecture [Newcomer and Lomow 2005], the great number of existing XML tools, its platform independence and many others.

Several works proposed XML Schemas or Document Type Definitions (DTDs) describing languages to represent DEVS models. [Wang and Lu 2002] develop a tool allowing the modeler to graphically build System Entity Structures [Zeigler et al. 2000], which can be used to capture the DEVS models structure. The graphical representation is then transformed into an XML document containing the model hierarchy.

Based on this premise, [Rhöl and Uhrmacher 2005] define XML Schemas for the DEVS formalism, including both coupled and atomic models. Model structure specification includes ports, couplings and state variables, and the emphasis is put on the use of XML data binding to automatically generate classes corresponding to XML models. However, the representation of the model dynamics is rather limited since each function is defined by a raw string, written in a specific programming language.

This problem of behavior representation is tackled in several works. One approach [Risco-Martín et al. 2007] is to restrict the different functions to simple constructs. For example, the transition functions are described by a set of conditional expressions, each being associated with new values for the state variables. A more general and powerful solution is to use some kind of pseudo-language, as is proposed in [Janoušek et al. 2006]. The advantage of using a simple language is that it can be easily represented in XML and transformed into source code in any programming language. Finally, [Mittal et al. 2007] use existing XML-based source code representation. In this case, the favored language is Java, but the authors evoke the possibility to use a more generic XML language to describe the model logic.

4.3. Limits and perspectives

The major problem that emerges from these previous works is the difficulty to standardize the representation of the model dynamics. Several solutions are possible, each with their own advantages and drawbacks.

The first idea is to use an XML representation of a specific programming language, e.g. JAVA Markup Language [Badros 2000]. This approach has the great advantage that many tools are available to manipulate and transform the source code and the XML representation. However, it induces a strong coupling between the model and a particular language, therefore restricting its deployment scope to a small set of simulators.

To enlarge this scope, a more generic representation could be employed, such as Metal [Lemos 1999] or Object Oriented XML (O2XML) [Wiharto and Stanski 2004]. These markup languages enable a representation of most object-oriented programming languages, by specifying a common denominator embracing all base features of these languages, in a .Net Common Language Specifications fashion. Thereby, an XML document can be transformed into source code in any object-oriented target language. Even though this solution might seems very attractive, it prevents the modeler from using certain language features or libraries that could be crucial. For example, one might want to generate pseudo-random numbers with the Stochastic Simulation in Java library [L'Ecuyer et al. 2002], or perform matrices operations using Boost.uBLAS. As a result, such a high-level language with a small set of features would reduce the universality of the standard, and make very hard the representation of legacy models, which are usually written in a specific programming language and take advantage of its functionalities and libraries.

To resolve this issue and be as permissive as possible, we could just let the modeler write the model functions in his favorite language, and delegate the task of interpreting them to simulators or proxies. However, that would imply developing an architecture where a model could be composed of several modules in multiple languages, this model being itself simulated by an application possibly written in yet another language. Even though this architecture is conceivable, using technologies such as JNI, RMI, CORBA or Web services, the complexity and the loss of performance would be huge.

To reach a compromise between genericity and flexibility, we define in this work our own XML representation of source code, by drawing inspiration from all these approaches. In addition, we propose some directions about several aspects that were left unspecified in previous works, such as parameterization and initialization of models, representation of their graphical attributes and standardization of trajectories.

4.4. Towards a unified representation

In this section, we propose a basis for a unified and standardized representation for each characteristic of DEVS models, as described above.

4.4.1. Model structure

Describing the structure of DEVS models is rather straight-forward, and is simply a translation from the set formalism into an XML representation. Figure 3 depicts our XML Schema arborescence for atomic models in classic DEVS.

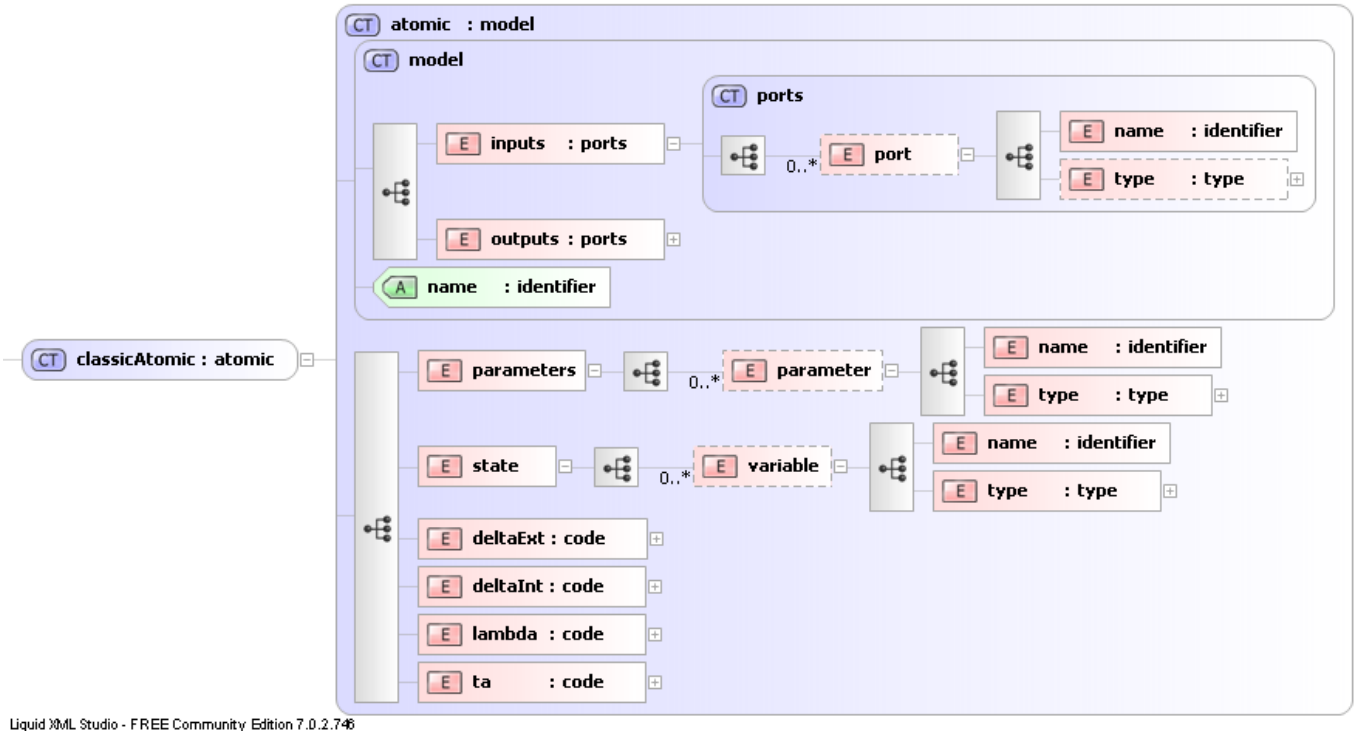

Figure 3: XML arborescence for classic atomic DEVS model

Input and output ports, parameters and state variables are characterized by a name and a type. To allow the deployment of the model on any simulator, this type should be as generic as possible; however, in real-world applications, the modeler might need to use certain types defined in particular language libraries. To satisfy these apparently contradictory requirements, we define three different kinds of types:

- intrisic types, which are natively supported by all object-oriented languages (e.g. integer, string, float, etc.)
- custom types, i.e. types that are defined by the modeler in a language independent manner
- language dependent types, which have to be bind to a particular type, for each targeted languages.

```
<state>
 <variable>
  <name>phase</name>
  <type kind="custom">procPhase</type>
 </variable>
 <variable>
  <name>buffer</name>
  <type kind="langDepend">Queue</type>
 </variable>
 <variable>
  <name>sigma</name>
  <type>double</type>
 </variable>
</state>
...
<type id="procPhase" xsi:type="enum">
 <value>busy</value>
 <value>idle</value>
</type>
<type id="Queue" xsi:type="langDepend">
 <impl lang="java">
  java.util.LinkedList<String>
 </impl>
 <impl lang="c++">
  std::queue<std::string>
 </impl>
</state>
```

Figure 4: Sample XML representation of state variables

Figure 4 shows an example use of these different types. In this sample, the state variable "sigma" is a simple floating-point number, whereas "phase" has a more complex type: it is an enumeration, described below as having two possible values, *busy* and *idle*. Finally, the variable "queue" has an abstract type, which we bind to specific types in Java and C++. Therefore, to make this model deployable on a simulator written in another language, the modeler simply has to specify what type can be used as a queue in this language. We can also imagine having repositories of abstract types along with their bindings in numerous programming languages, to allow their reusability in several models.

The XML Schema for coupled DEVS models has no major additional subtleties. It includes ports, components and coupling declarations, as well as a tie-breaking function in the case of classic DEVS. Components are described by their name and the type of model they instantiate. The couplings are pairs of ports, if necessary qualified by their component's name. Finally, the *Select* function is described by ordered sets indicating priorities among particular group of components.

4.4.2. Model dynamics

Section 3.3 identifies several issues concerning the specification of atomic DEVS models behavior. These problems boil down to the following question: How can we describe application logic in a language-independent manner? Several works have been done to provide an XML representation of source code. Some of them are oriented towards a sole programming language, e.g. JAVAML [Badros 2000], PascalML [McArthur et al. 2002] or CppML [Mamas and Kontogiannis 2000], while others propose vocabularies that contain only the common denominator between all object-oriented languages [Mamas and Kontogiannis 2000] [Wiharto and Stanski 2004].

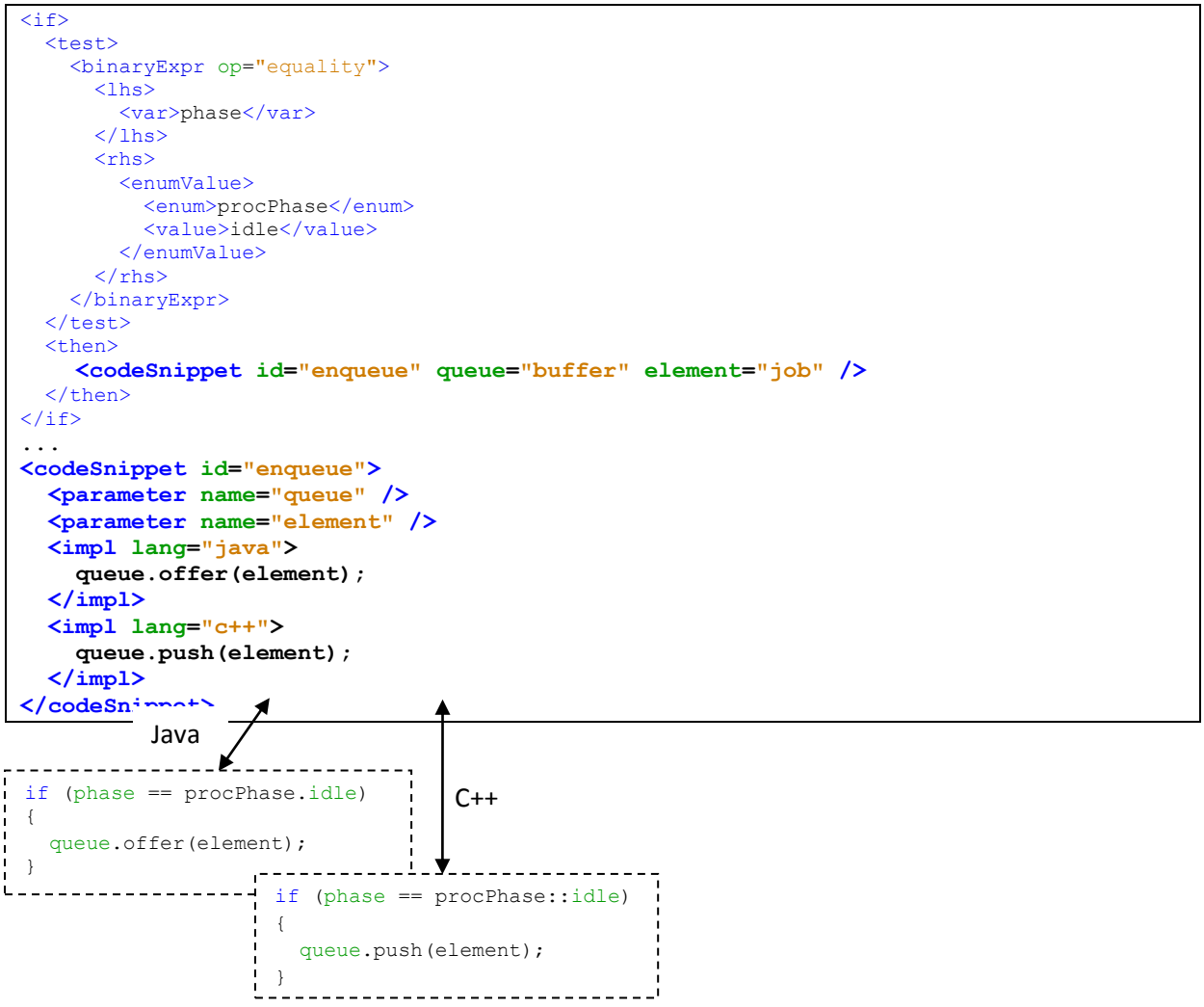

Figure 5: Representation of model dynamics using a semi-generic language

None of these works provide a framework being at the same time as generic as possible, and open to extension with language dependent features. Therefore, we propose a compromise between both approaches, by drawing inspiration from the concept of weaving used in Aspect Oriented Programming [Kiczales et al. 1997], or more simply from the C/C++ macros.

The main idea is to use a pseudo-language such as O2XML so that the major part of the code can be reused across different simulators, while letting the possibility to include language specific snippets. This concept is illustrated in Figure 5, where generic programming constructs are represented in XML, and library uses are abstracted and bind to specific implementations in several languages.

This semi-generic language can also be used to create custom types (i.e. classes, enumerations), which become usable in the variables and ports declarations.

This approach has two major advantages:

- allowing the use of programming language features and libraries.
- permitting the representation of legacy models written for a specific simulator.

By limiting the language dependent code snippets to a minimum, we facilitate the migration of models between platforms and thereby enhance their portability and reusability.

4.4.3. Parameterization and initialization

The description of DEVS models as described above is not sufficient to produce an executable specification. Indeed, models can have parameters, which need to be set prior to simulation, and their initial state must be specified in some way.

Figure 6 depicts how a model, regarded as a template, can be instantiated into several parameterized model depending on the values assigned to each parameters. After that, the state must be initialized to obtain a ready-to-simulate model. A simulator can then be fed with this executable model, along with an input trajectory if need be, and produce the output of the simulation.

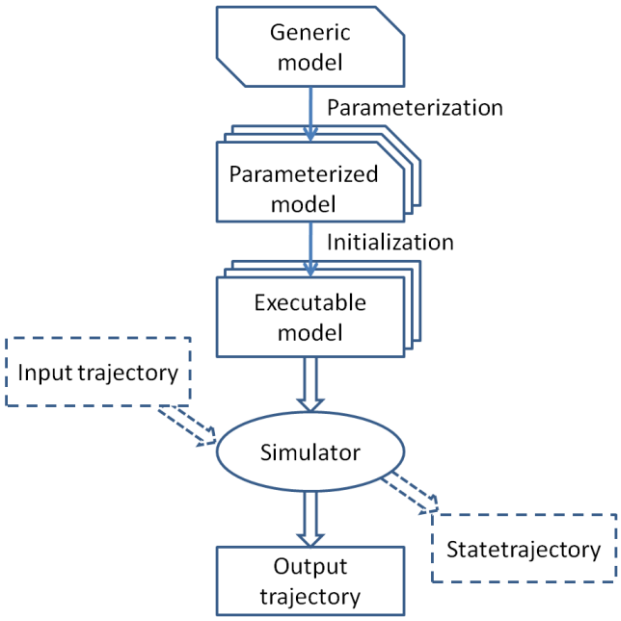

Figure 6: Parameterization and initialization of a model

To perform these operations, two approaches are possible depending on the degree of freedom required. The simplest way is to restrict parameterization and initialization to pairs of variable identifiers and values. However, these operations can sometimes be more complex than mere assignments, e.g. when dealing with file manipulation or random number generation. Therefore, our proposition let these two activities be described in a similar fashion as the model dynamics, using the not-so-generic source code representation outlined in the previous section.

Consequently, the modeler needs to define two functions for each atomic model in the hierarchy: one to set the parameters values, and the other one to initialize the state. Ideally, these functions should be kept separate from the model definition, so that it can be easily reused with different scenarios.

In order to simulate a DEVS model, four specifications are needed: structure, dynamics, parameterization and initialization. Given these specifications and if required an input trajectory, a simulator can produce results representing the model behavior.

4.4.4. Model behavior

The behavior of a model is normally fully represented by its output trajectory. However, it is common to desire its state trajectory and, in the case of coupled models, the behavior of its components as well. These two aspects can be described by set of values annotated with a timestamp. Figure 7 shows a sample state trajectory of a coupled model, namely the *pipeSimple* defined in [Zeigler et al. 2000].

The XML schema for this representation of trajectory can be deduced from the model structure: each variable or port become an XML tag accepting a value in accordance with its type. Another solution would be to have a tag *variable* with an attribute containing the name of the variable.

In the case of variables of complex type, the value at a given time can be obtained by serializing the object from memory,
as done when exchanging data between web services. This supposes providing an XML Schema describing the structure of the object, for later use (e.g. in a visualization tool). For custom types, this schema can be deduced from the type definition made in the model specification; for language dependent types, the schema can often be automatically generated from the source code.

```
<behavior id="pipeSimple">
  <state>
    <snapshot time="0">
      <component id="p0">
        <phase>idle</phase>
        <queue />
        <sigma>infinity</sigma>
      </component>
      <!-- other components -->
    </snapshot>
    <snapshot time="3">
      <component id="p0">
        <phase>busy</phase>
        <queue>
          <elem>job1</elem>
        </queue>
        <sigma>10</sigma>
       </component>
       <!-- and so on -->
```

Figure 7: Sample state trajectory

This XML representation of trajectories has the great advantage that it can be easily processed by a machine while remaining human readable; however, due to the potentially huge size of data generated, it might be necessary to switch to a more compact representation, such the eXternal Data Representation standard [Vangheluwe et al. 2001].

4.4.5. Presentation

Finally, a modeler should be able to save and load its models to a repository without losing their graphical attributes (position on the layout, size and so on). However, these pieces of information are very dependent on the graphical user interface employed, and therefore should be uncoupled from the model specification, and stored in separate files, in the style of Cascading Style Sheets used in web development.

5. Conclusion

In this paper we proposed three possible approaches for providing interoperability between DEVS components. The first one, simulator-based interoperability is to standardize exchanges between simulators and have been successfully experimented in DEVS/SOA. The two others focus on models instead of simulators. In the on-line approach, we deploy models as web services so that they can be used remotely by any simulator, using model

transformations for generating these services and the code for invoking them. In the off-line approach, we go one step further in the code generation process by generating a complete operational copy of the models on the local simulator. To make possible these two model-driven strategies, we use a platform and language-independent description of DEVS models done with the DEVS Markup Language.

The propositions made here are rather prospective; however, some works have been done towards the realization of the on-line model-based architecture, which promise interesting follow-ups. The off-line solution, involving multiple complex model transformations, need to be looked into more thoroughly to demonstrate its feasibility, and is a work in progress.

REFERENCES

[Abiteboul et al. 1999] S. Abiteboul, P. Buneman, and D. Suciu. "Data on the Web: From Relational to Semistructured Data and XML". Morgan Kaufmann, 1999.

[Badros 2000] G.J. Badros. "JavaML: a Markup Language for Java Source Code". Proceedings of the 9th International World Wide Web Conference on Computer Networks. 2000

[Brutzman et al. 2002] D. Brutzman et al. "Extensible Modeling and Simulation Framework (XMSF) - Challenges for Web-Based Modeling and Simulation". Technical Challenges Workshop, Strategic Opportunities Symposium 2002

[L'Ecuyer et al. 2002] P. L'Ecuyer, L. Meliani, J. Vaucher. "SSJ: A Framework for Stochastic Simulation in Java". Proceedings of the 2002 Winter Simulation Conference

[Fishwick 2002] P.A. Fishwick. "Using XML for Simulation Modeling". Proceedings of the 2002 Winter Simulation Conference

[Janoušek et al. 2006] V. Janoušek, P. Polášek, P. Slavíček. "Towards DEVS Meta Language". ISC 2006 Proceedings

[Keith at al. 2000] B. Keith et al., "Service-Based Software: the Future for Flexible Software", (PDF) in Seventh Asia-Pacific Software Engineering Conference, 2000. APSEC 2000. Proceedings: Seventh Asia-Pacific Software Engineering Conference, 2000, pp. 214-221. doi:10.1109/APSEC.2000.896702

[Kiczales et al. 1997] G. Kiczales et al. "Aspect-Oriented Programming". Proceedings of the European Conference on Object-Oriented Programming. 1997

[Lemos 1999] M. Lemos. "MetaL: An XML-based Meta-Programming Language". http://www.meta-language.net/

[Mamas and Kontogiannis 2000] E. Mamas, K. Kontogiannis. "Towards Portable Source Code Representations Using XML". Proceedings of the Seventh Working Conference on Reverse Engineering. 2000

[McArthur et al. 2002] G. McArthur, J. Mylopoulos, S.K. Ng. "An Extensible Tool for Source Code Representations Using XML". Proceedings of the Ninth Working Conference on Reverse Engineering. 2002

[Mittal et al. 2007] S. Mittal, J.-L. Risco-Martín, B.P. Zeigler. "DEVSML: automating DEVS execution over SOA towards transparent simulators". Proceedings of SpringSim 2007

[Newcomer and Lomow 2005] E Newcomer, G Lomow. "Understanding Soa With Web Services". Addison-Wesley Professional. 2005

[Rhöl and Uhrmacher 2005] M. Rhöl, A.M. Uhrmacher. "Flexible Integration of XML into Modeling and Simulation Systems". Proceedings of the 2005 Winter Simulation Conference

[Risco-Martín et al. 2007] J.-L. Risco-Martín, S. Mittal, M. A. López-Peña, J. M. de la Cruz. "A W3C XML schema for DEVS scenarios". Proceedings of SpringSim 2007

[Schmidt 2006] D.C. Schmidt, Model-Driven Engineering - guest editor'introduction, IEEE Computer, February 2006 (Vol. 39, No. 2) pp. 25-31

[Seo 2009] C. Seo, "Interoperability between DEVS Simulators using Service Oriented Architecture and DEVS Namespace", Ph.D. dissertation, Electrical and Computer Engineering Dept., University of Arizona, Spring 2009

[Touraille et al. 2009] L. Touraille, M.K. Traore, D.R.C. Hill. "A Mark-up Language for the Storage, Retrieval, Sharing and Interoperability of DEVS Models". Proceedings of the 2009 ACM/SCS Spring Simulation Multiconference, SpringSim 2009, San Diego, CA, USA, March 22-27, 2009

[Vangheluwe et al. 2001] H. Vangheluwe, J. de Lara, J.-S. Bolduc, E. Posse, "DEVS Standardization: some thoughts". Winter Simulation Conference 2001

[Wang and Lu 2002] Y.H. Wang, Y.C. Lu. "An XML-based DEVS Modeling Tool to Enhance Simulation Interoperability". Proceeding 14th European Simulation Symposium, 2002

[Wiharto and Stanski 2004] M. Wiharto, P. Stanski. "An Architecture for Retargeting Application Logic to Multiple Component Types in Multiple Languages". Fifth Australasian Workshop on Software and System Architectures, 2004

[Zeigler et al. 2000] B.P. Zeigler, H. Praehofer, T.G. Kim. "Theory of Modeling and Simulation: Integrating Discrete and Continuous Complex Dynamic Systems". 2nd Edition, Academic press 2000, ISBN 0-12-778455-1